Code QMC=Chem (Chimie-Quantique) Vectorisation efficace sur processeurs scalaires

Anthony Scemama

21/09/2017

Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse, France

Le QMC en quelques mots

Résolution stochastique de l'équation de Schrödinger pour les électrons (les noyaux sont fixes):

$$
\begin{aligned}
E &= \frac{\int \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_N) \hat{H} \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_N) \, \mathrm{d}\mathbf{r}_1 \ldots \mathrm{d}\mathbf{r}_N}{\int \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_N) \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_N) \, \mathrm{d}\mathbf{r}_1 \ldots \mathrm{d}\mathbf{r}_N} \\
&\sim \frac{1}{M} \sum_M \frac{\hat{H}\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)}{\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)}, \text{ échantillonné avec } (\Psi \times \Phi)
\end{aligned}
$$

$\hat{H}$: Opérateur Hamiltonien

E: Énergie

$\mathbf{r}_1, \ldots, \mathbf{r}_N$: Cordonnées des électrons

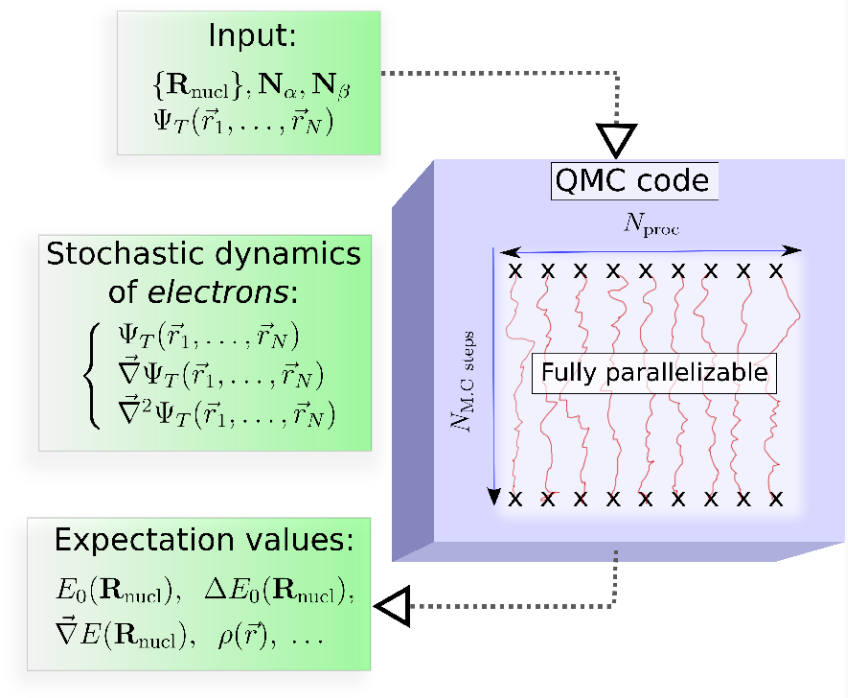
Φ: Fonction d'onde exacte

Ψ: Fonction d'onde d'essai

Le QMC en quelques mots

- Marcheur: Vecteur de $\mathbb{R}^{3N}$ qui contient les coordonnées des électrons
- Diffusion + dérive des marcheurs avec un processus de mort/naissance pour engendrer la $3N$-densité ($\Psi \times \Phi$)

- À chaque pas, on calcule $E_{\text{loc}} = \frac{\hat{H}\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_N)}{\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_N)}$
- L'énergie est la moyenne de toutes les E_{loc} calculées.
- Parallélisme : on distribue des populations de trajectoires sur différents CPUs.



Rendre le code efficace

Pas de synchronisation nécessaire, donc la performance croît linéairement avec le nombre de CPUs. L'optimisation mono-cœur est *cruciale*.

À chaque pas il faut calculer :

- $\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)$
- $\vec{\nabla}\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)$
- $\Delta\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)$

Deux points chauds

1. Produit d'une matrice dense $A^{N \times N_b}$ par 5 vecteurs creux $\{B_1, \ldots, B_5\}^{N_b \times N}$ qui ont les éléments non-nuls aux mêmes indices.
2. Beaucoup d'inversion de petites matrices pour le calcul des dérivées

On peut utiliser la simple précision : les erreurs se compenseront dans le calcul des moyennes

Difficultés

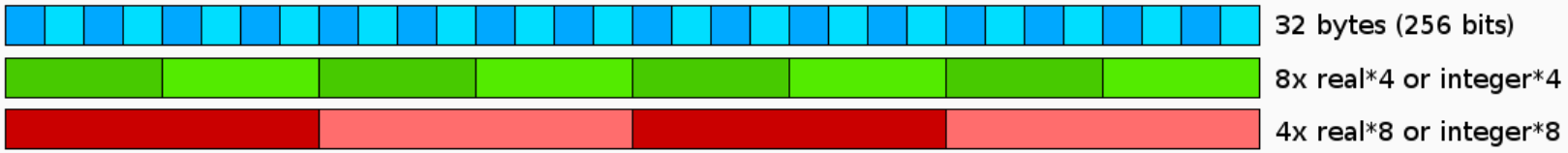
- L'algèbre linaire creuse réduit l'intensité arithmétique : memory-bound
- Les matrices sont petites : $N \sim 50$, $N_b \sim 200$: la vectorisation automatique n'a pas le temps de "chauffer"
- La simple précision double la taille des vecteurs et rend la vectorisation encore plus difficile

Rappels sur la vectorisation

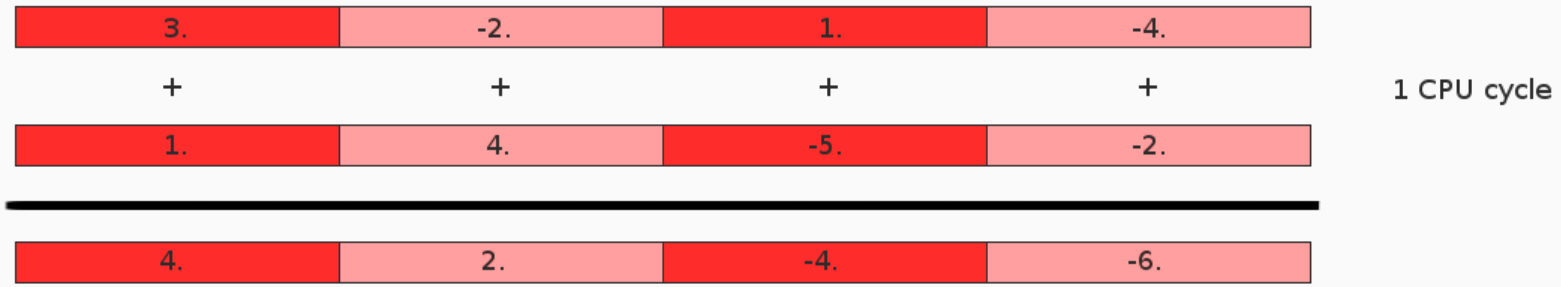
Jeux d'instructions sur x86:

MMX	: Int,	64 bit	AVX	: Int, FP,	256 bit
SSE4.2	: Int,FP,	128 bit	AVX-512	: Int, FP,	512 bit

Parallélisme SIMD (Single Instruction Multiple Data):



Addition vectorielle AVX en double précision:



Rappels sur la vectorisation

Contraintes

- Les données doivent arriver suffisamment vite
  - Caches, ré-utilisabilité
  - Éviter les struct/types dérivés
  - Patterns prévisibles (réguliers)
- Les éléments doivent être consécutifs en mémoire
  - 1^{er} indice en Fortran, dernier indice en C/C++
  - Gather/scatter pour les accès avec stride > 1
- Les vecteurs doivent être correctement alignés en mémoire
  - `!DIR$ ATTRIBUTES ALIGN`
  - `-align array32byte`

Meilleur accès aux données : transposition

On stocke des copies transposées de tableaux $\Longrightarrow$
accès "stride 1"

```
do j=1,n
 do i=1,n
  C(i,j) = 0.d0
  do k=1,n
!     Mauvais accès à A
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
  end do
 end do
end do
```

Meilleur accès aux données : transposition

On stocke des copies transposées de tableaux $\Longrightarrow$
accès "stride 1"

```
do j=1,n
 do i=1,n
  C(i,j) = 0.d0
  do k=1,n
!     Bon accès à A_tranp
      C(i,j) = C(i,j) + A_tranp(k,i) * B(k,j)
  end do
 end do
end do
```

Augmentation de l'intensité arithmétique

Unroll and jam

- On déroule la boucle extérieure
- On fusionne les lignes
- Moins de store $\Longrightarrow$ moins d'accès à la mémoire

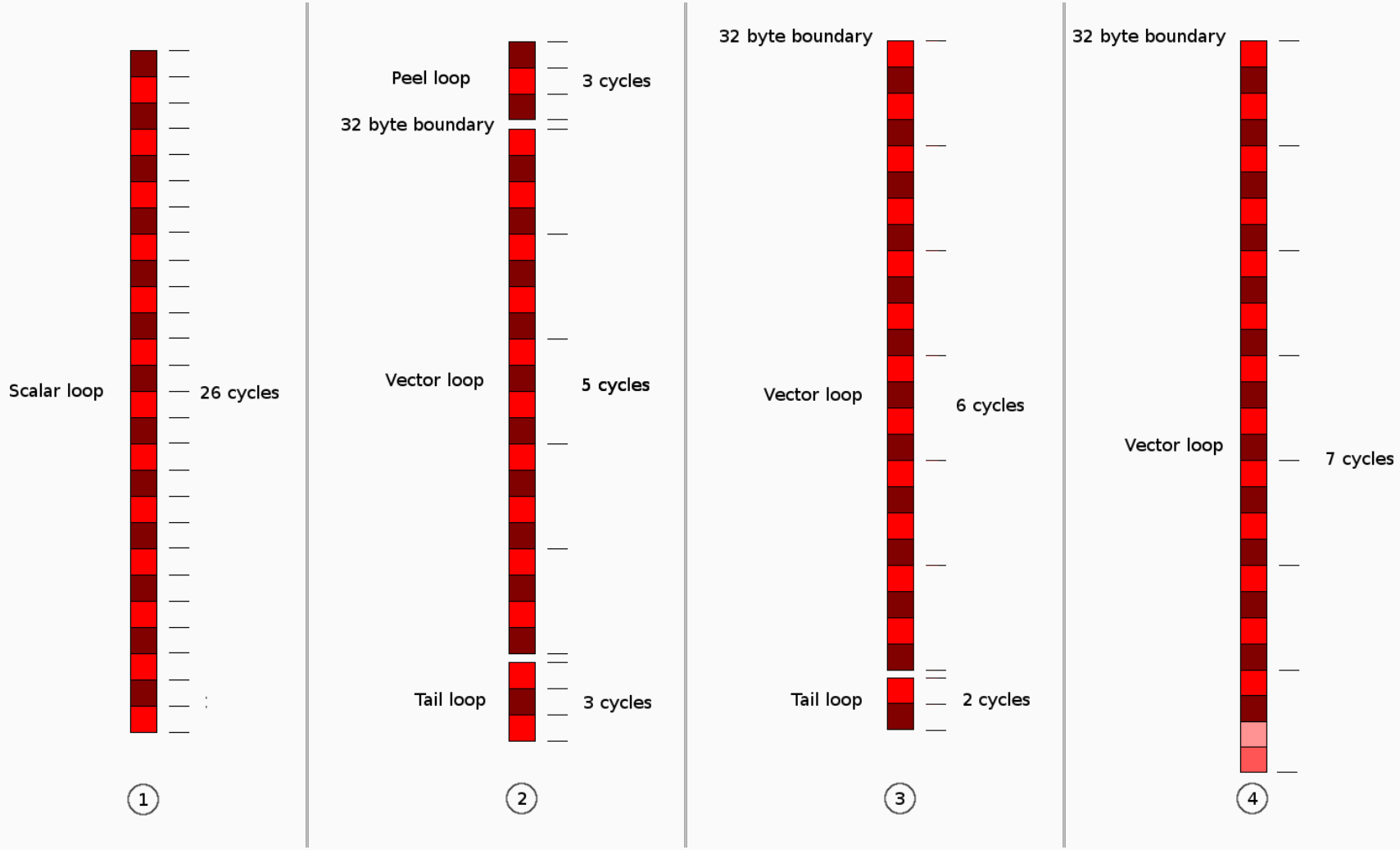
```
do j=1,n
 do k=1,n,4
  do i=1,n
    C(i,j)=C(i,j) + A(i,k  )*B(k  ,j) + A(i,k+1)*B(k+1,j) &
                  + A(i,k+2)*B(k+2,j) + A(i,k+3)*B(k+3,j)
  end do
 end do
end do
```

Vectorisation automatique

1 boucle (Fortran) génère 4 boucles (binaire)

- Une version scalaire
- Une peel loop : 1^{ers} éléments jusqu'à la frontière d'alignement
- Une version vectorielle
- Une tail loop : éléments restants

Vectorisation automatique



Optimisations

Allocation alignée sur 32 octets :

```
double precision, allocatable :: A(:), B(:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A, B
```

On peut indiquer au compilateur que les tableaux sont alignés :

```
!DIR$ VECTOR ALIGNED
do i=1,n
  A(i) = A(i) + B(i)
end do
```

$\Longrightarrow$ suppression de la peel loop

Optimisations

Si n est un multiple de l'alignement (4 DP pour AVX), on peut supprimer la tail loop $\Longrightarrow$ padding:

```
m = n/4
n_4 = m*4
if (n_4 < n) n_4 = (m+1)*4
allocate ( A(n_4), B(n_4) )
do i=1,n,4
  !DIR$ VECTOR ALIGNED
  !DIR$ VECTOR ALWAYS
  do k=0,3
    A(i+k) = A(i+k) + B(i+k)
  end do
end do
```

On choisit 8 plutôt que 4 : on déroule 2$\times$ la boucle sur i.

Optimisations: Tableaux 2D

```
m = n/8
n_8 = m*8
if (n_8 < n) n_8 = (m+1)*8
allocate ( A(n_8,n), B(n_8,n) )
do j=1,n
  do i=1,n,8
    !DIR$ VECTOR ALIGNED
    !DIR$ VECTOR ALWAYS
    do k=0,7
      A(i+k,j) = A(i+k,j) + B(i+k,j)
    end do
  end do
end do
```

Toutes les colonnes sont alignées $\Longrightarrow$ code 100% vectorisé.

Exemple: matrice des distances

```
do j=1,n
 do i=1,j ! <-------
  dist1(i,j)= (X(i,1)-X(j,1)) * (X(i,1)-X(j,1)) + &
              (X(i,2)-X(j,2)) * (X(i,2)-X(j,2)) + &
              (X(i,3)-X(j,3)) * (X(i,3)-X(j,3))
 end do
end do
```

```
do j=1,n
 do i=j+1,n
  dist1(i,j) = dist1(j,i)
 end do
end do
```

$t(n = 133) = 13.0\mu s, 3.0$ GFlops/s

$t(n = 4125) = 95.4ms, 0.44$ GFlops/s

Exemple: matrice des distances

```
do j=1,n
 do i=1,n ! <------- 2 fois plus de flops
  dist1(i,j)= (X(i,1)-X(j,1)) * (X(i,1)-X(j,1)) + &
              (X(i,2)-X(j,2)) * (X(i,2)-X(j,2)) + &
              (X(i,3)-X(j,3)) * (X(i,3)-X(j,3))
 end do
end do
```

$t(n = 133) = 10.3\mu\text{s}, 8.2$ GFlops/s $\times 1.12$

$t(n = 4125) = 15.7\text{ms}, 5.4$ GFlops/s $\times 5.75$

Exemple: matrice des distances

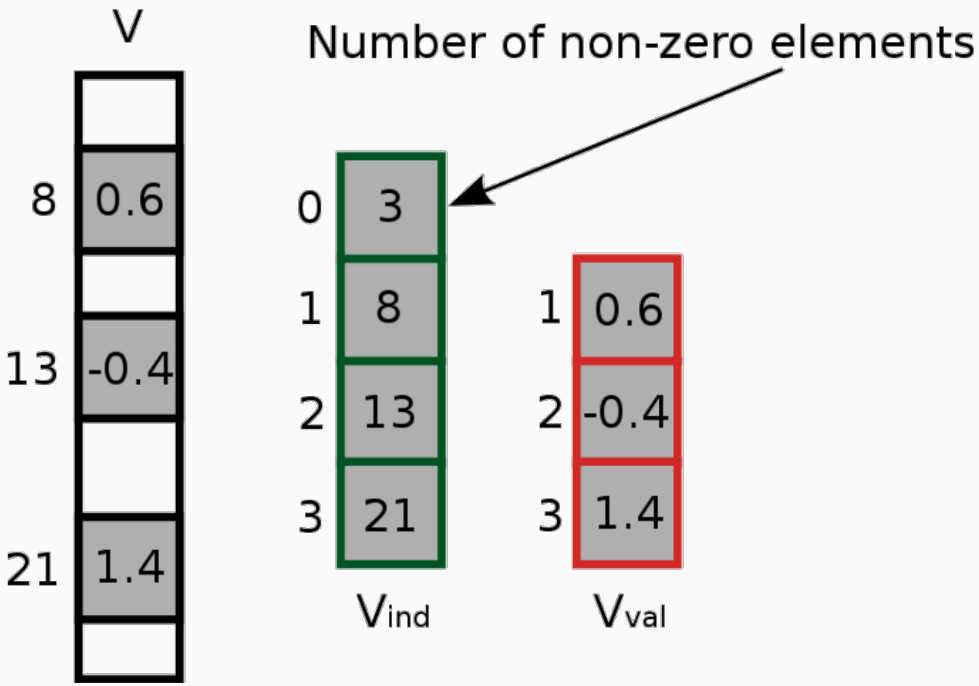
```
do j=1,n
 do i=1,n,8
  !DIR$ VECTOR ALIGNED
  !DIR$ VECTOR ALWAYS
  do k=0,7
   dist1(i+k,j)= (X(i+k,1)-X(j,1)) * (X(i+k,1)-X(j,1)) + &
                 (X(i+k,2)-X(j,2)) * (X(i+k,2)-X(j,2)) + &
                 (X(i+k,3)-X(j,3)) * (X(i+k,3)-X(j,3))
  end do
 end do
end do
```

$t(n = 133) = 7.2\mu\text{s}, 12.1$ GFlops/s $\times 1.80$

$t(n = 4125) = 15.7\text{ms}, 7.5$ GFlops/s $\times 6.15$

Stockage

- Chaque colonne a au moins 1 élément non-nul
- Le nombre d'éléments non-nuls est relativement constant
- Stockage LOL (list of lists) : accès direct à la colonne



Produit matrice-vecteur

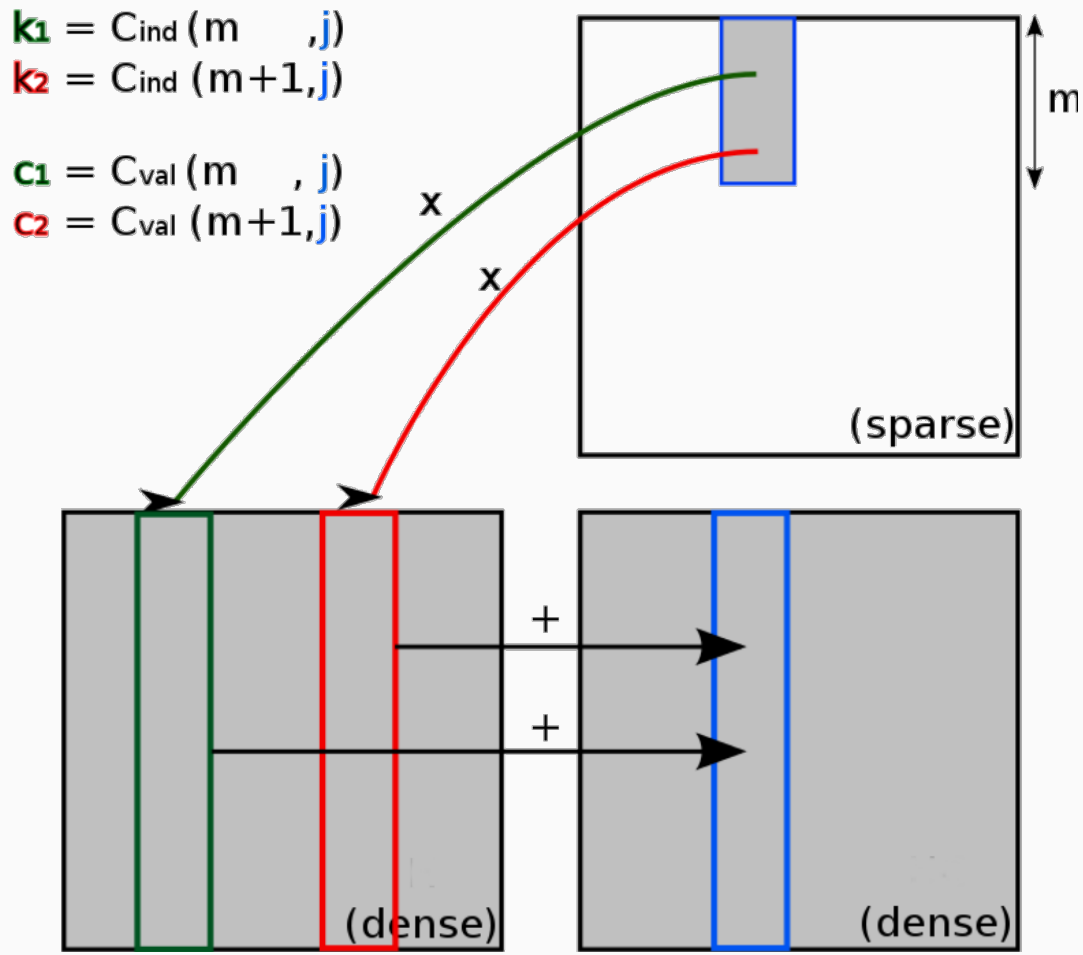


Table 1: Single core performance (GFlops/s), % peak in parenthesis, Measured on Intel Xeon E31240, 3.30GHz (52.8 GFlops/s SP, 26.4 GFlops/s DP).

N_{elec}	N_{basis}	Matrix inversion	Matrix products	Overall (1 core)
158	404	6.3 (24%)	26.6 (50%)	8.8 (23%)
434	963	14.0 (53%)	33.1 (63%)	11.8 (33%)
434	2934	14.0 (53%)	33.6 (64%)	13.7 (38%)
1056	2370	17.9 (67%)	30.6 (58%)	15.2 (49%)
1731	3892	17.8 (67%)	28.2 (53%)	16.2 (55%)

Génération de routines spécialisées par un programme

- $N < 5 = 0$: Algorithme en N!
- Pour $N \geq 5$, utilisation de Sherman-Morrison:

$$(A + uv^\dagger)^{-1} = A^{-1} - \frac{A^{-1}uv^\dagger A^{-1}}{1 + v^\dagger A^{-1}u}$$

- Templates dépendant de $N \mod 4 \in \{0, 1, 2, 3\}$
- Si $N \mod 4 > 0$, tail loops ré-écrites à la main sous forme vectorisée

Exemple: 4×4

```
 character*(7) :: irp_here = 'invert4'                                  ! TOOLS/invert.irp.f: 160
double precision, intent(inout) :: a (LDA,na)                           ! TOOLS/invert.irp.f: 162
integer, intent(in)             :: LDA                                  ! TOOLS/invert.irp.f: 163
integer, intent(in)             :: na                                   ! TOOLS/invert.irp.f: 164
double precision, intent(inout) :: det_l                                ! TOOLS/invert.irp.f: 165
double precision :: b(4,4)                                              ! TOOLS/invert.irp.f: 166
integer :: i,j                                                          ! TOOLS/invert.irp.f: 168
double precision :: f                                                   ! TOOLS/invert.irp.f: 169
!DIR$ ATTRIBUTES ALIGN : 32 :: b                                        ! TOOLS/invert.irp.f: 167
det_l =  a(1,1)*(a(2,2)*(a(3,3)*a(4,4)-a(3,4)*a(4,3))  &
                -a(2,3)*(a(3,2)*a(4,4)-a(3,4)*a(4,2))  &
                +a(2,4)*(a(3,2)*a(4,3)-a(3,3)*a(4,2))) &
       -a(1,2)*(a(2,1)*(a(3,3)*a(4,4)-a(3,4)*a(4,3))  &
                -a(2,3)*(a(3,1)*a(4,4)-a(3,4)*a(4,1))  &
                +a(2,4)*(a(3,1)*a(4,3)-a(3,3)*a(4,1))) &
       +a(1,3)*(a(2,1)*(a(3,2)*a(4,4)-a(3,4)*a(4,2))  &
                -a(2,2)*(a(3,1)*a(4,4)-a(3,4)*a(4,1))  &
                +a(2,4)*(a(3,1)*a(4,2)-a(3,2)*a(4,1))) &
       -a(1,4)*(a(2,1)*(a(3,2)*a(4,3)-a(3,3)*a(4,2))  &
                -a(2,2)*(a(3,1)*a(4,3)-a(3,3)*a(4,1))  &
                +a(2,3)*(a(3,1)*a(4,2)-a(3,2)*a(4,1))) ! TOOLS/invert.irp.f: 170
do i=1,4                                                                ! TOOLS/invert.irp.f: 182
 b(1,i) = a(1,i)                                                        ! TOOLS/invert.irp.f: 183
 b(2,i) = a(2,i)                                                        ! TOOLS/invert.irp.f: 184
 b(3,i) = a(3,i)                                                        ! TOOLS/invert.irp.f: 185
 b(4,i) = a(4,i)                                                        ! TOOLS/invert.irp.f: 186
enddo                                                                   ! TOOLS/invert.irp.f: 187
a(1,1) =  b(2,2)*(b(3,3)*b(4,4)-b(3,4)*b(4,3))-b(2,3)*(b(3,2)*b(4,4)-b(3,4)*b(4,2))+b(2,4)*(b(3,2)*b(4,3)-b(3,3)*b(4,2))
a(2,1) = -b(2,1)*(b(3,3)*b(4,4)-b(3,4)*b(4,3))+b(2,3)*(b(3,1)*b(4,4)-b(3,4)*b(4,1))-b(2,4)*(b(3,1)*b(4,3)-b(3,3)*b(4,1))
a(3,1) =  b(2,1)*(b(3,2)*b(4,4)-b(3,4)*b(4,2))-b(2,2)*(b(3,1)*b(4,4)-b(3,4)*b(4,1))+b(2,4)*(b(3,1)*b(4,2)-b(3,2)*b(4,1))
a(4,1) = -b(2,1)*(b(3,2)*b(4,3)-b(3,3)*b(4,2))+b(2,2)*(b(3,1)*b(4,3)-b(3,3)*b(4,1))-b(2,3)*(b(3,1)*b(4,2)-b(3,2)*b(4,1))
a(1,2) = -b(1,2)*(b(3,3)*b(4,4)-b(3,4)*b(4,3))+b(1,3)*(b(3,2)*b(4,4)-b(3,4)*b(4,2))-b(1,4)*(b(3,2)*b(4,3)-b(3,3)*b(4,2))
a(2,2) =  b(1,1)*(b(3,3)*b(4,4)-b(3,4)*b(4,3))-b(1,3)*(b(3,1)*b(4,4)-b(3,4)*b(4,1))+b(1,4)*(b(3,1)*b(4,3)-b(3,3)*b(4,1))
a(3,2) = -b(1,1)*(b(3,2)*b(4,4)-b(3,4)*b(4,2))+b(1,2)*(b(3,1)*b(4,4)-b(3,4)*b(4,1))-b(1,4)*(b(3,1)*b(4,2)-b(3,2)*b(4,1))
a(4,2) =  b(1,1)*(b(3,2)*b(4,3)-b(3,3)*b(4,2))-b(1,2)*(b(3,1)*b(4,3)-b(3,3)*b(4,1))+b(1,3)*(b(3,1)*b(4,2)-b(3,2)*b(4,1))
a(1,3) =  b(1,2)*(b(2,3)*b(4,4)-b(2,4)*b(4,3))-b(1,3)*(b(2,2)*b(4,4)-b(2,4)*b(4,2))+b(1,4)*(b(2,2)*b(4,3)-b(2,3)*b(4,2))
a(2,3) = -b(1,1)*(b(2,3)*b(4,4)-b(2,4)*b(4,3))+b(1,3)*(b(2,1)*b(4,4)-b(2,4)*b(4,1))-b(1,4)*(b(2,1)*b(4,3)-b(2,3)*b(4,1))
a(3,3) =  b(1,1)*(b(2,2)*b(4,4)-b(2,4)*b(4,2))-b(1,2)*(b(2,1)*b(4,4)-b(2,4)*b(4,1))+b(1,4)*(b(2,1)*b(4,2)-b(2,2)*b(4,1))
a(4,3) = -b(1,1)*(b(2,2)*b(4,3)-b(2,3)*b(4,2))+b(1,2)*(b(2,1)*b(4,3)-b(2,3)*b(4,1))-b(1,3)*(b(2,1)*b(4,2)-b(2,2)*b(4,1))
a(1,4) = -b(1,2)*(b(2,3)*b(3,4)-b(2,4)*b(3,3))+b(1,3)*(b(2,2)*b(3,4)-b(2,4)*b(3,2))-b(1,4)*(b(2,2)*b(3,3)-b(2,3)*b(3,2))
a(2,4) =  b(1,1)*(b(2,3)*b(3,4)-b(2,4)*b(3,3))-b(1,3)*(b(2,1)*b(3,4)-b(2,4)*b(3,1))+b(1,4)*(b(2,1)*b(3,3)-b(2,3)*b(3,1))
a(3,4) = -b(1,1)*(b(2,2)*b(3,4)-b(2,4)*b(3,2))+b(1,2)*(b(2,1)*b(3,4)-b(2,4)*b(3,1))-b(1,4)*(b(2,1)*b(3,2)-b(2,2)*b(3,1))
a(4,4) =  b(1,1)*(b(2,2)*b(3,3)-b(2,3)*b(3,2))-b(1,2)*(b(2,1)*b(3,3)-b(2,3)*b(3,1))+b(1,3)*(b(2,1)*b(3,2)-b(2,2)*b(3,1))
end                                                                     ! TOOLS/invert.irp.f: 209
```

Template pour $N \mod 4 = 0$

```
subroutine det_update$n(n,LDS,m,l,S,S_inv,d)
  implicit none
  integer, intent(in)   :: n,LDS  ! Dimension of the vector
  real, intent(in)      :: m($n)  ! New vector
  integer, intent(in)   :: l      ! New position in S
  real,intent(inout)    :: S(LDS,$n)     ! Slater matrix
  real*8,intent(inout)  :: S_inv(LDS,$n) ! Inverse Slater ma
  real*8,intent(inout)  :: d             ! Det(S)
  real*8                :: u($n), z($n), w($n), lambda, d_in
  !DIR$ ATTRIBUTES ALIGN : 32 :: z, w, u
  !DIR$ ASSUME_ALIGNED S : 32
  !DIR$ ASSUME_ALIGNED S_inv : 32
  !DIR$ ASSUME (mod(LDS,4) == 0)
  !DIR$ ASSUME (LDS >= $n)
  ...
```

Template pour $N \mod 4 = 0$

```
zj = 0.d0
!DIR$ VECTOR ALIGNED
!DIR$ SIMD REDUCTION(+:zj) NOVECREMAINDER
do i=1,$n-1,4
  zj = zj + S_inv(i,l)*u(i) + S_inv(i+1,l)*u(i+1)  &
          + S_inv(i+2,l)*u(i+2) + S_inv(i+3,l)*u(i+3)
end do
d_inv = 1.d0/d
d = d+zj
lambda = d*d_inv
if (dabs(lambda) < 1.d-3) then
  d = 0.d0
  return
endif
...
```

Template pour $N \mod 4 = 0$

```
!DIR$ VECTOR ALIGNED
do j=1,$n,4
 zj  = 0.d0 ; zj1 = 0.d0 ; zj2 = 0.d0 ; zj3 = 0.d0
 !DIR$ VECTOR ALIGNED
 !DIR$ SIMD REDUCTION(+:zj,zj1,zj2,zj3) NOVECREMAINDER
 do i=1,$n
  zj  = zj  + S_inv(i,j  )*u(i)
  zj1 = zj1 + S_inv(i,j+1)*u(i)
  zj2 = zj2 + S_inv(i,j+2)*u(i)
  zj3 = zj3 + S_inv(i,j+3)*u(i)
 end do
 z(j  ) = zj  ; z(j+1) = zj1
 z(j+2) = zj2 ; z(j+3) = zj3
end do
...
```

Template pour $N \mod 4 = 0$

```
!DIR$ SIMD FIRSTPRIVATE(d_inv) NOVECREMAINDER
do i=1,$n
  w(i) = S_inv(i,l)*d_inv
  S(i,l) = m(i)
end do
...
```

1 division DP : 20 cycles

1 multiplication DP : 0.25 cycles

Template pour $N \mod 4 = 0$

```
  do i=1,$n,4
   zj  = z(i  ) ; zj1 = z(i+1)
   zj2 = z(i+2) ; zj3 = z(i+3)
   !DIR$ VECTOR ALIGNED
   !DIR$ SIMD FIRSTPRIVATE(lambda,zj,zj1,zj2,zj3) NOVECREMA
   do j=1,$n
    S_inv(j,i  ) = S_inv(j,i  )*lambda - w(j)*zj
    S_inv(j,i+1) = S_inv(j,i+1)*lambda - w(j)*zj1
    S_inv(j,i+2) = S_inv(j,i+2)*lambda - w(j)*zj2
    S_inv(j,i+3) = S_inv(j,i+3)*lambda - w(j)*zj3
   end do
  end do
end
```

Template pour $N \mod 4 = 2$

```
! zj = 0.d0
! !DIR$ VECTOR ALIGNED
! !DIR$ SIMD REDUCTION(+:zj) NOVECREMAINDER
! do i=1,$n-1,4
  do i=1,$n-2,4
!   zj = zj + S_inv(i,l)*u(i) + S_inv(i+1,l)*u(i+1)  &
!           + S_inv(i+2,l)*u(i+2) + S_inv(i+3,l)*u(i+3)
! end do
  i=$n-1
  zj = zj + S_inv(i,l)*u(i) + S_inv(i+1,l)*u(i+1)

  ...
```

Template pour $N \mod 2 = 0$

```
! !DIR$ VECTOR ALIGNED
! do j=1,$n,4
  do j=1,$n-2,4
!  zj  = 0.d0 ; zj1 = 0.d0 ; zj2 = 0.d0 ; zj3 = 0.d0
!  !DIR$ VECTOR ALIGNED
!  !DIR$ SIMD REDUCTION(+:zj,zj1,zj2,zj3) NOVECREMAINDER
!  do i=1,$n
   do i=1,$n-2
!   zj  = zj  + S_inv(i,j  )*u(i)
!   zj1 = zj1 + S_inv(i,j+1)*u(i)
!   zj2 = zj2 + S_inv(i,j+2)*u(i)
!   zj3 = zj3 + S_inv(i,j+3)*u(i)
!  end do
!  z(j  ) = zj  ; z(j+1) = zj1
!  z(j+2) = zj2 ; z(j+3) = zj3
   z(j  )=zj + S_inv($n-1,j  )*u($n-1) + S_inv($n,j  )*u($n)
   z(j+1)=zj1+ S_inv($n-1,j+1)*u($n-1) + S_inv($n,j+1)*u($n)
   z(j+2)=zj2+ S_inv($n-1,j+2)*u($n-1) + S_inv($n,j+2)*u($n)
   z(j+3)=zj3+ S_inv($n-1,j+3)*u($n-1) + S_inv($n,j+3)*u($n)
! end do
 ...
```

Template pour $N \mod 2 = 0$

```
j=$n-1
zj  = 0.d0
zj1 = 0.d0
!DIR$ VECTOR ALIGNED
!DIR$ SIMD REDUCTION(+:zj,zj1) NOVECREMAINDER
do i=1,$n-2
 zj  = zj  + S_inv(i,j  )*u(i)
 zj1 = zj1 + S_inv(i,j+1)*u(i)
end do
z(j  ) = zj + S_inv($n-1,j  )*u($n-1) &
            + S_inv($n  ,j  )*u($n  )
z(j+1) = zj1+ S_inv($n-1,j+1)*u($n-1) &
            + S_inv($n  ,j+1)*u($n  )
...
```

Performances

Inversion de matrices 15×15

Algorithme	Machine	$\times 10^6 s^{-1}$	$\times 10^6 Tflops^{-1}$
$\mathcal{O}(N^3)$	Xeon E5-1620	1.49	6.65
(MKL)	4c@3.5GHz, AVX2		
	KNL 7210	6.12	2.30
	64c@1.3GHz, AVX512		
$\mathcal{O}(N^2)$	Xeon E3-1271	5.1	88.5
(QMC=Chem)	4c@3.6GHz, AVX2		

$13\times$ plus rapide $\Longrightarrow$ efficacité comparable à MKL.